NAME

DBD::DBM - a DBI driver for DBM & MLDBM files

SYNOPSIS

```
use DBI;
$dbh = DBI->connect('dbi:DBM:'); # defaults to SDBM_File
$dbh = DBI->connect('DBI:DBM(RaiseError=1):'); # defaults to SDBM_File
$dbh = DBI->connect('dbi:DBM:dbm_type=DB_File'); # defaults to DB_File
$dbh = DBI->connect('dbi:DBM:dbm_mldbm=Storable'); # MLDBM with SDBM_File

# or
$dbh = DBI->connect('dbi:DBM:', undef, undef);
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
f_ext => '.db/r',
f_dir => '/path/to/dbfiles/',
f_lockfile => '.lck',
dbm_type => 'BerkeleyDB',
dbm_mldbm => 'FreezeThaw',
dbm_store_metadata => 1,
dbm_berkeley_flags => {
'-Cachesize' => 1000, # set a ::Hash flag
},
});
```

and other variations on *connect()* as shown in the DBI docs, DBD::File metadata and "Metadata" shown below.

Use standard DBI prepare, execute, fetch, placeholders, etc., see "QUICK START" for an example.

DESCRIPTION

DBD::DBM is a database management system that works right out of the box. If you have a standard installation of Perl and DBI you can begin creating, accessing, and modifying simple database tables without any further modules. You can add other modules (e.g., SQL::Statement, DB_File etc) for improved functionality.

The module uses a DBM file storage layer. DBM file storage is common on many platforms and files can be created with it in many programming languages using different APIs. That means, in addition to creating files with DBI/SQL, you can also use DBI/SQL to access and modify files created by other DBM modules and programs and vice versa. **Note** that in those cases it might be necessary to use a common subset of the provided features.

DBM files are stored in binary format optimized for quick retrieval when using a key field. That optimization can be used advantageously to make DBD::DBM SQL operations that use key fields very fast. There are several different "flavors" of DBM which use different storage formats supported by perl modules such as SDBM_File and MLDBM. This module supports all of the flavors that perl supports and, when used with MLDBM, supports tables with any number of columns and insertion of Perl objects into tables.

DBD::DBM has been tested with the following DBM types: SDBM_File, NDBM_File, ODBM_File, GDBM_File, DB_File, BerkeleyDB. Each type was tested both with and without MLDBM and with the Data::Dumper, Storable, FreezeThaw, YAML and JSON serializers using the DBI::SQL::Nano or the SQL::Statement engines.

QUICK START

DBD::DBM operates like all other DBD drivers - it's basic syntax and operation is specified by DBI. If you're not familiar with DBI, you should start by reading DBI and the documents it points to and then come back and read this file. If you are familiar with DBI, you already know most of what you need to know to operate this module. Just jump in and create a test script something like the one shown below.

You should be aware that there are several options for the SQL engine underlying DBD::DBM, see "Supported SQL syntax". There are also many options for DBM support, see especially the section on "Adding multi-column support with MLDBM".

But here's a sample to get you started.

```
use DBI;
my $dbh = DBI->connect('dbi:DBM:');
$dbh->{RaiseError} = 1;
for my $sql( split /;\n+/,"
CREATE TABLE user ( user_name TEXT, phone TEXT );
INSERT INTO user VALUES ('Fred Bloggs','233-7777');
INSERT INTO user VALUES ('Sanjay Patel','777-3333');
INSERT INTO user VALUES ('Junk','xxx-xxxx');
DELETE FROM user WHERE user_name = 'Junk';
UPDATE user SET phone = '999-4444' WHERE user_name = 'Sanjay Patel';
SELECT * FROM user
"){
my $sth = $dbh->prepare($sql);
$sth->execute;
$sth->dump_results if $sth->{NUM_OF_FIELDS};
}
$dbh->disconnect;
```

USAGE

This section will explain some usage cases in more detail. To get an overview about the available attributes, see "Metadata".

Specifying Files and Directories

DBD::DBM will automatically supply an appropriate file extension for the type of DBM you are using. For example, if you use SDBM_File, a table called "fruit" will be stored in two files called "fruit.pag" and "fruit.dir". You should **never** specify the file extensions in your SQL statements.

DBD::DBM recognizes following default extensions for following types:

.pag/r

Chosen for dbm_type `SDBM_File`, `ODBM_File` and `NDBM_File` when an implementation is detected which wraps `-ldbm` for `NDBM_File` (e.g. Solaris, AIX, ...).

For those types, the `.dir` extension is recognized, too (for being deleted when dropping a table).

.db/r

Chosen for dbm_type `NDBM_File` when an implementation is detected which wraps BerkeleyDB 1.x for `NDBM_File` (typically BSD's, Darwin).

`GDBM_File`, `DB_File` and `BerkeleyDB` don't usually use a file extension.

If your DBM type uses an extension other than one of the recognized types of extensions, you should set the *f_ext* attribute to the extension **and** file a bug report as described in DBI with the name of the implementation and extension so we can add it to DBD::DBM. Thanks in advance for that :-).

```
$dbh = DBI->connect('dbi:DBM:f_ext=.db'); # .db extension is used
$dbh = DBI->connect('dbi:DBM:f_ext='); # no extension is used

# or
$dbh->{f_ext}='.db'; # global setting
$dbh->{f_meta}->{'qux'}->{f_ext}='.db'; # setting for table 'qux'
```

By default files are assumed to be in the current working directory. To use other directories specify the *f_dir* attribute in either the connect string or by setting the database handle attribute.

For example, this will look for the file /foo/bar/fruit (or /foo/bar/fruit.pag for DBM types that use that extension)

```
my $dbh = DBI->connect('dbi:DBM:f_dir=/foo/bar');
# and this will too:
my $dbh = DBI->connect('dbi:DBM:');
$dbh->{f_dir} = '/foo/bar';
# but this is recommended
my $dbh = DBI->connect('dbi:DBM:', undef, undef, { f_dir => '/foo/bar' } );

# now you can do
my $ary = $dbh->selectall_arrayref(q{ SELECT x FROM fruit });
```

You can also use delimited identifiers to specify paths directly in SQL statements. This looks in the same place as the two examples above but without setting *f_dir*:

```
my $dbh = DBI->connect('dbi:DBM:');
my $ary = $dbh->selectall_arrayref(q{
SELECT x FROM "/foo/bar/fruit"
});
```

You can also tell DBD::DBM to use a specified path for a specific table:

```
$dbh->{dbm_tables}->{f}->{file} = q(/foo/bar/fruit);
```

Please be aware that you cannot specify this during connection.

If you have SQL::Statement installed, you can use table aliases:

```
my $dbh = DBI->connect('dbi:DBM:');
my $ary = $dbh->selectall_arrayref(q{
SELECT f.x FROM "/foo/bar/fruit" AS f
});
```

See the "GOTCHAS AND WARNINGS" for using DROP on tables.

Table locking and *flock()*

Table locking is accomplished using a lockfile which has the same basename as the table's file but with the file extension '.lck' (or a lockfile extension that you supply, see below). This lock file is created with the table during a CREATE and removed during a DROP. Every time the table itself is opened, the lockfile is *flocked()*. For SELECT, this is a shared lock. For all other operations, it is an exclusive lock (except when you specify something different using the *f_lock* attribute).

Since the locking depends on *flock()*, it only works on operating systems that support *flock()*. In cases where *flock()* is not implemented, DBD::DBM will simply behave as if the *flock()* had occurred although no actual locking will happen. Read the documentation for *flock()* for more information.

Even on those systems that do support *flock()*, locking is only advisory - as is always the case with *flock()*. This means that if another program tries to access the table file while DBD::DBM has the table locked, that other program will *succeed* at opening unless it is also using flock on the '.lck' file. As a result DBD::DBM locking only really applies to other programs using DBD::DBM or other program written to cooperate with DBD::DBM locking.

Specifying the DBM type

Each "flavor" of DBM stores its files in a different format and has different capabilities and limitations. See AnyDBM_File for a comparison of DBM types.

By default, DBD::DBM uses the `SDBM_File` type of storage since `SDBM_File` comes with Perl itself. If you have other types of DBM storage available, you can use any of them with DBD::DBM. It is strongly recommended to use at least `DB_File`, because `SDBM_File` has quirks and limitations and `ODBM_file`, `NDBM_File` and `GDBM_File` are not always available.

You can specify the DBM type using the *dbm_type* attribute which can be set in the connection string or with `$dbh->{dbm_type}` and `$dbh->{f_meta}->{$table_name}->{type}` for per-table settings in cases where a single script is accessing more than one kind of DBM file.

In the connection string, just set `dbm_type=TYPENAME` where `TYPENAME` is any DBM type such as GDBM_File, DB_File, etc. Do *not* use MLDBM as your *dbm_type* as that is set differently, see below.

```
my $dbh=DBI->connect('dbi:DBM:'); # uses the default SDBM_File
my $dbh=DBI->connect('dbi:DBM:dbm_type=GDBM_File'); # uses the GDBM_File

# You can also use $dbh->{dbm_type} to set the DBM type for the connection:
$dbh->{dbm_type} = 'DB_File'; # set the global DBM type
print $dbh->{dbm_type}; # display the global DBM type
```

If you have several tables in your script that use different DBM types, you can use the $dbh->{dbm_tables} hash to store different settings for the various tables. You can even use this to perform joins on files that have completely different storage mechanisms.

```
# sets global default of GDBM_File
my $dbh->('dbi:DBM:type=GDBM_File');

# overrides the global setting, but only for the tables called
# I<foo> and I<bar>
my $dbh->{f_meta}->{foo}->{dbm_type} = 'DB_File';
my $dbh->{f_meta}->{bar}->{dbm_type} = 'BerkeleyDB';

# prints the dbm_type for the table "foo"
print $dbh->{f_meta}->{foo}->{dbm_type};
```

Note that you must change the *dbm_type* of a table before you access it for first time.

Adding multi-column support with MLDBM

Most of the DBM types only support two columns and even if it would support more, DBD::DBM would only use two. However a CPAN module called MLDBM overcomes this limitation by allowing more than two columns. MLDBM does this by serializing the data - basically it puts a reference to an array into the second column. It can also put almost any kind of Perl object or even **Perl coderefs**in to columns.

If you want more than two columns, you **must** install MLDBM. It's available for many platforms and is easy to install.

MLDBM is by default distributed with three serializers - Data::Dumper, Storable, and FreezeThaw. Data::Dumper is the default and Storable is the fastest. MLDBM can also make use of user-defined serialization methods or other serialization modules (e.g. YAML::MLDBM or MLDBM::Serializer::JSON. You select the serializer using the *dbm_mldbm* attribute.

Some examples:

```
$dbh=DBI->connect('dbi:DBM:dbm_mldbm=Storable'); # use MLDBM with Storable
$dbh=DBI->connect(
'dbi:DBM:dbm_mldbm=MySerializer' # use MLDBM with a user defined module
);
$dbh=DBI->connect('dbi::dbm:', undef,
undef, { dbm_mldbm => 'YAML' }); # use 3rd party serializer
$dbh->{dbm_mldbm} = 'YAML'; # same as above
print $dbh->{dbm_mldbm} # show the MLDBM serializer
$dbh->{f_meta}->{foo}->{dbm_mldbm}='Data::Dumper'; # set Data::Dumper for table "foo"
print $dbh->{f_meta}->{foo}->{mldbm}; # show serializer for table "foo"
```

MLDBM works on top of other DBM modules so you can also set a DBM type along with setting dbm_mldbm. The examples above would default to using SDBM_File with MLDBM. If you wanted GDBM_File instead, here's how:

```
# uses DB_File with MLDBM and Storable
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
dbm_type => 'DB_File',
dbm_mldbm => 'Storable',
});
```

SDBM_File, the default *dbm_type* is quite limited, so if you are going to use MLDBM, you should probably use a different type, see AnyDBM_File.

See below for some "GOTCHAS AND WARNINGS" about MLDBM.

Support for Berkeley DB

The Berkeley DB storage type is supported through two different Perl modules - DB_File (which supports only features in old versions of Berkeley DB) and BerkeleyDB (which supports all versions). DBD::DBM supports specifying either "DB_File" or "BerkeleyDB" as a *dbm_type*, with or without MLDBM support.

The "BerkeleyDB" dbm_type is experimental and it's interface is likely to change. It currently defaults to BerkeleyDB::Hash and does not currently support ::Btree or ::Recno.

With BerkeleyDB, you can specify initialization flags by setting them in your script like this:

```
use BerkeleyDB;
my $env = new BerkeleyDB::Env -Home => $dir; # and/or other Env flags
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
dbm_type => 'BerkeleyDB',
dbm_mldbm => 'Storable',
dbm_berkeley_flags => {
'DB_CREATE' => DB_CREATE, # pass in constants
'DB_RDONLY' => DB_RDONLY, # pass in constants
'-Cachesize' => 1000, # set a ::Hash flag
'-Env' => $env, # pass in an environment
},
});
```

Do *not* set the -Flags or -Filename flags as those are determined and overwritten by the SQL (e.g. -Flags => DB_RDONLY is set automatically when you issue a SELECT statement).

Time has not permitted us to provide support in this release of DBD::DBM for further Berkeley DB features such as transactions, concurrency, locking, etc. We will be working on these in the future and would value suggestions, patches, etc.

See DB_File and BerkeleyDB for further details.

Optimizing the use of key fields

Most "flavors" of DBM have only two physical columns (but can contain multiple logical columns as explained above in "Adding multi-column support with MLDBM"). They work similarly to a Perl hash with the first column serving as the key. Like a Perl hash, DBM files permit you to do quick lookups by specifying the key and thus avoid looping through all records (supported by DBI::SQL::Nano only). Also like a Perl hash, the keys must be unique. It is impossible to create two records with the same key. To put this more simply and in SQL terms, the key column functions as the *PRIMARY KEY* or UNIQUE INDEX.

In DBD::DBM, you can take advantage of the speed of keyed lookups by using DBI::SQL::Nano and a WHERE clause with a single equal comparison on the key field. For example, the following SQL statements are optimized for keyed lookup:

```
CREATE TABLE user ( user_name TEXT, phone TEXT);
INSERT INTO user VALUES ('Fred Bloggs','233-7777');
# ... many more inserts
SELECT phone FROM user WHERE user_name='Fred Bloggs';
```

The "user_name" column is the key column since it is the first column. The SELECT statement uses the key column in a single equal comparison - "user_name='Fred Bloggs'" - so the search will find it very quickly without having to loop through all the names which were inserted into the table.

In contrast, these searches on the same table are not optimized:

```
1. SELECT phone FROM user WHERE user_name < 'Fred';
2. SELECT user_name FROM user WHERE phone = '233-7777';
```

In #1, the operation uses a less-than (<) comparison rather than an equals comparison, so it will not be optimized for key searching. In #2, the key field "user_name" is not specified in the WHERE clause, and therefore the search will need to loop through all rows to find the requested row(s).

Note that the underlying DBM storage needs to loop over all *key/value* pairs when the optimized fetch is used. SQL::Statement has a massively improved where clause evaluation which costs around 15% of the evaluation in DBI::SQL::Nano - combined with the loop in the DBM storage the speed improvement isn't so impressive.

Even if lookups are faster by around 50%, DBI::SQL::Nano and SQL::Statement can benefit from the key field optimizations on updating and deleting rows - and here the improved where clause evaluation of SQL::Statement might beat DBI::SQL::Nano every time the where clause contains not only the key field (or more than one).

Supported SQL syntax

DBD::DBM uses a subset of SQL. The robustness of that subset depends on what other modules you have installed. Both options support basic SQL operations including CREATE TABLE, DROP TABLE, INSERT, DELETE, UPDATE, and SELECT.

Option #1: By default, this module inherits its SQL support from DBI::SQL::Nano that comes with DBI. Nano is, as its name implies, a *very* small SQL engine. Although limited in scope, it is faster than option #2 for some operations (especially single *primary key* lookups). See DBI::SQL::Nano for a description of the SQL it supports and comparisons of it with option #2.

Option #2: If y ou install the pure Perl CPAN module SQL::Statement, DBD::DBM will use it instead of Nano. This adds support for table aliases, functions, joins, and much more. If you're going to use DBD::DBM for anything other than very simple tables and queries, you should install SQL::Statement. You don't have to change DBD::DBM or your scripts in any way, simply installing SQL::Statement will give you the more robust SQL capabilities without breaking scripts written for DBI::SQL::Nano. See SQL::Statement for a description of the SQL it supports.

To find out which SQL module is working in a given script, you can use the *dbm_versions()* method or, if you don't need the full output and version numbers, just do this:

```
print $dbh->{sql_handler}, "\n";
```

That will print out either "SQL::Statement" or "DBI::SQL::Nano"

Baring the section about optimized access to the DBM storage in mind, comparing the benefits of both engines:

```
# DBI::SQL::Nano is faster
$sth = $dbh->prepare( "update foo set value='new' where key=15" );
$sth->execute();
$sth = $dbh->prepare( "delete from foo where key=27" );
$sth->execute();
$sth = $dbh->prepare( "select * from foo where key='abc'" );

# SQL::Statement might faster (depending on DB size)
$sth = $dbh->prepare( "update foo set value='new' where key=?" );
$sth->execute(15);
$sth = $dbh->prepare( "update foo set value=? where key=15" );
$sth->execute('new');
$sth = $dbh->prepare( "delete from foo where key=?" );
$sth->execute(27);

# SQL::Statement is faster
$sth = $dbh->prepare( "update foo set value='new' where value='old'" );
$sth->execute();
# must be expressed using "where key = 15 or key = 27 or key = 42 or key = 'abc'"
# in DBI::SQL::Nano
$sth = $dbh->prepare( "delete from foo where key in (15,27,42,'abc')" );
$sth->execute();
# must be expressed using "where key > 10 and key < 90" in DBI::SQL::Nano
$sth = $dbh->prepare( "select * from foo where key between (10,90)" );
$sth->execute();

# only SQL::Statement can handle
$sth->prepare( "select * from foo,bar where foo.name = bar.name" );
$sth->execute();
$sth->prepare( "insert into foo values ( 1, 'foo' ), ( 2, 'bar' )" );
$sth->execute();
```

Specifying Column Names

DBM files don't have a standard way to store column names. DBD::DBM gets around this issue with a DBD::DBM specific way of storing the column names. **If y ou are working only with DBD::DBM and not using files created by or accessed with other DBM programs, you can ignore this section.**

DBD::DBM stores column names as a row in the file with the key *_metadata 0*. So this code

```
my $dbh = DBI->connect('dbi:DBM:');
$dbh->do("CREATE TABLE baz (foo CHAR(10), bar INTEGER)");
$dbh->do("INSERT INTO baz (foo,bar) VALUES ('zippy',1)");
```

Will create a file that has a structure something like this:

```
_metadata \0 | <dbd_metadata><schema></schema><col_names>foo,bar</col_names></dbd_metadata>
zippy | 1
```

The next time you access this table with DBD::DBM, it will treat the *_metadata 0*ro w as a header rather than as data and will pull the column names from there. However, if you access the file with something other than DBD::DBM, the row will be treated as a regular data row.

If you do not want the column names stored as a data row in the table you can set the *dbm_store_metadata* attribute to 0.

```
my $dbh = DBI->connect('dbi:DBM:', undef, undef, { dbm_store_metadata => 0 });

# or
```

```
$dbh->{dbm_store_metadata} = 0;

# or for per-table setting
$dbh->{f_meta}->{qux}->{dbm_store_metadata} = 0;
```

By default, DBD::DBM assumes that you have two columns named "k" and "v" (short for "key" and "value"). So if you have *dbm_store_metadata* set to 1 and you want to use alternate column names, you need to specify the column names like this:

```
my $dbh = DBI->connect('dbi:DBM:', undef, undef, {
dbm_store_metadata => 0,
dbm_cols => [ qw(foo bar) ],
});

# or
$dbh->{dbm_store_metadata} = 0;
$dbh->{dbm_cols} = 'foo,bar';

# or to set the column names on per-table basis, do this:
# sets the column names only for table "qux"
$dbh->{f_meta}->{qux}->{dbm_store_metadata} = 0;
$dbh->{f_meta}->{qux}->{col_names} = [qw(foo bar)];
```

If you have a file that was created by another DBM program or created with *dbm_store_metadata* set to zero and you want to convert it to using DBD::DBM column name storage, just use one of the methods above to name the columns but *without* specifying *dbm_store_metadata* as zero. You only have to do that once - thereafter you can get by without setting either *dbm_store_metadata* or setting *dbm_cols* because the names will be stored in the file.

DBI database handle attributes

Metadata

Statement handle (sth) attributes and methods

Most statement handle attributes such as NAME, NUM_OF_FIELDS, etc. are available only after an execute. The same is true of `$sth`->rows which is available after the execute but does *not* require a fetch.

Driver handle (dbh) attributes

It is not supported anymore to use dbm-attributes without the dbm_-prefix. Currently, if an DBD::DBM private attribute is accessed without an underscore in it's name, dbm_ is prepended to that attribute and it's processed further. If the resulting attribute name is invalid, an error is thrown.

dbm_cols

Contains a comma separated list of column names or an array reference to the column names.

dbm_type

Contains the DBM storage type. Currently known supported type are `ODBM_File`, `NDBM_File`, `SDBM_File`, `GDBM_File`, `DB_File` and `BerkeleyDB`. It is not recommended to use one of the first three types - even if `SDBM_File` is the most commonly available *dbm_type*.

dbm_mldbm

Contains the serializer for DBM storage (value column). Requires the CPAN module MLDBM installed. Currently known supported serializers are:

Data::Dumper
    Default serializer. Deployed with Perl core.

Storable
: Faster serializer. Deployed with Perl core.

FreezeThaw
: Pure Perl serializer, requires FreezeThaw to be installed.

YAML
: Portable serializer (between languages but not architectures). Requires YAML::MLDBM installation.

JSON
: Portable, fast serializer (between languages but not architectures). Requires MLDBM::Serializer::JSON installation.

dbm_store_metadata

Boolean value which determines if the metadata in DBM is stored or not.

dbm_berkeley_flags

Hash reference with additional flags for BerkeleyDB::Hash instantiation.

dbm_version

Readonly attribute containing the version of DBD::DBM.

f_meta

In addition to the attributes DBD::File recognizes, DBD::DBM knows about the (public) attributes `col_names` (**Note** not *dbm_cols* here!), `dbm_type`, `dbm_mldbm`, `dbm_store_metadata` and `dbm_berkeley_flags`. As in DBD::File, there are undocumented, internal attributes in DBD::DBM. Be very careful when modifying attributes you do not know; the consequence might a destroyed or corrupted table.

dbm_tables

This attribute provides restricted access to the table meta data. See f_meta and "f_meta" in DBD::File for attribute details.

dbm_tables is a tied hash providing the internal table names as keys (accessing unknown tables might create an entry) and their meta data as another tied hash. The table meta storage is obtained via the`get_table_meta` method from the table implemen tation (see DBD::File::Developers). Attribute setting and getting within the table meta data is handled via the methods `set_table_meta_attr` and `get_table_meta_attr`.

Following attributes are no longer handled by DBD::DBM:

dbm_ext

This attribute is silently mapped to DBD::File's attribute *f_ext*. Later versions of DBI might show a depreciated warning when this attribute is used and eventually it will be removed.

dbm_lockfile

This attribute is silently mapped to DBD::File's attribute *f_lockfile*. Later versions of DBI might show a depreciated warning when this attribute is used and eventually it will be removed.

DBI database handle methods

The `$dbh->`*dbm_versions()* method

The private method *dbm_versions()* returns a summary of what other modules are being used at any given time. DBD::DBM can work with or without many other modules - it can use either SQL::Statement or DBI::SQL::Nano as its SQL engine, it can be run with DBI or DBI::PurePerl, it can use many kinds of DBM modules, and many kinds of serializers when run with MLDBM. The *dbm_versions()* method reports all of that and more.

```
print $dbh->dbm_versions; # displays global settings
print $dbh->dbm_versions($table_name); # displays per table settings
```

An important thing to note about this method is that when it called with no arguments, it displays the *global* settings. If you override these by setting per-table attributes, these will *not* be shown unless you specify a table name as an argument to the method call.

Storing Objects

If you are using MLDBM, you can use DBD::DBM to take advantage of its serializing abilities to serialize any Perl object that MLDBM can handle. To store objects in columns, you should (but don't absolutely need to) declare it as a column of type BLOB (the type is *currently* ignored by the SQL engine, but it's good form).

EXTENSIBILITY

`SQL::Statement`
: Improved SQL engine compared to the built-in DBI::SQL::Nano - see "Supported SQL syntax".

`DB_File`
: Berkeley DB version 1. This database library is available on many systems without additional installation and most systems are supported.

`GDBM_File`
: Simple dbm type (comparable to `DB_File`) under the GNU license. Typically not available (or requires extra installation) on non-GNU operating systems.

`BerkeleyDB`
: Berkeley DB version up to v4 (and maybe higher) - requires additional installation but is easier than GDBM_File on non-GNU systems.

db4 comes with a many tools which allow repairing and migrating databases. This is the **recommended** dbm type for production use.

`MLDBM`
: Serializer wrapper to support more than one column for the files. Comes with serializers using `Data::Dumper` `FreezeThaw` and `Storable`.

`YAML::MLDBM`
: Additional serializer for MLDBM. YAML is very portable between languages.

`MLDBM::Serializer::JSON`
: Additional serializer for MLDBM. JSON is very portable between languages, probably more than YAML.

GOTCHAS AND WARNINGS

Using the SQL DROP command will remove any file that has the name specified in the command with either '.pag' and '.dir', '.db' or your {f_ext} appended to it. So this be dangerous if you aren't sure what file it refers to:

```
 $dbh->do(qq{DROP TABLE "/path/to/any/file"});
```

Each DBM type has limitations. SDBM_File, for example, can only store values of less than 1,000 characters. *You* as the script author must ensure that you don't exceed those bounds. If you try to insert a value that is larger than DBM can store, the results will be unpredictable. See the documentation for whatever DBM you are using for details.

Different DBM implementations return records in different orders. That means that you *should not* rely on the order of records unless you use an ORDER BY statement.

DBM data files are platform-specific. To move them from one platform to another, you'll need to do something along the lines of dumping your data to CSV on platform #1 and then dumping from CSV to DBM on platform #2. DBD::AnyData and DBD::CSV can help with that. There may also be DBM conversion tools for your platforms which would probably be quicker.

When using MLDBM, there is a very powerful serializer - it will allow you to store Perl code or objects in database columns. When these get de-serialized, they may be eval'ed - in other words MLDBM (or actually Data::Dumper when used by MLDBM) may take the values and try to execute them in Perl. Obviously, this can present dangers, so if you do not know what is in a file, be careful before you access it with MLDBM turned on!

See the entire section on "Table locking and *flock()*" for gotchas and warnings about the use of *flock()*.

BUGS AND LIMITATIONS

This module uses hash interfaces of two column file databases. While none of supported SQL engines have support for indices, the following statements really do the same (even if they mean something completely different) for each dbm type which lacks `EXISTS` support:

```
$sth->do( "insert into foo values (1, 'hello')" );

# this statement does ...
$sth->do( "update foo set v='world' where k=1" );
# ... the same as this statement
$sth->do( "insert into foo values (1, 'world')" );
```

This is considered to be a bug and might change in a future release.

Known affected dbm types are `ODBM_File` and `NDBM_File`. We highly recommended you use a more modern dbm type such as `DB_File`.

GETTING HELP, MAKING SUGGESTIONS, AND REPORTING BUGS

If you need help installing or using DBD::DBM, please write to the DBI users mailing list at dbi-users@perl.org or to the comp.lang.perl.modules newsgroup on usenet. I cannot always answer every question quickly but there are many on the mailing list or in the newsgroup who can.

DBD developers for DBD's which rely on DBD::File or DBD::DBM or use one of them as an example are suggested to join the DBI developers mailing list at dbi-dev@perl.org and strongly encouraged to join our IRC channel at <irc://irc.perl.org/dbi>.

If you have suggestions, ideas for improvements, or bugs to report, please report a bug as described in DBI. Do not mail any of the authors directly, you might not get an answer.

When reporting bugs, please send the output of `$dbh->`dbm_versions($table) for a table that exhibits the bug and as small a sample as you can make of the code that produces the bug. And of course, patches are welcome, too :-).

If you need enhancements quickly, you can get commercial support as described at <http://dbi.perl.org/support/> or you can contact Jens Rehsack at rehsack@cpan.org for commercial support in Germany.

Please don't bother Jochen Wiedmann or Jeff Zucker for support - they handed over further maintenance to H.Merijn Brand and Jens Rehsack.

ACKNOWLEDGEMENTS

Many, many thanks to Tim Bunce for prodding me to write this, and for copious, wise, and patient suggestions all along the way. (Jeff Zucker)

I send my thanks and acknowledgements to H.Merijn Brand for his initial refactoring of DBD::File and his strong and ongoing support of SQL::Statement. Without him, the current progress would never have been made. And I have to name Martin J. Evans for each laugh (and correction) of all those funny word creations I (as non-native speaker) made to the documentation. And - of course - I have to thank all those unnamed contributors and testers from the Perl community. (Jens Rehsack)

AUTHOR AND COPYRIGHT

This module is written by Jeff Zucker < jzucker AT cpan.org >, who also maintained it till 2007. After that, in 2010, Jens Rehsack & H.Merijn Brand took over maintenance.

SEE ALSO

DBI, SQL::Statement, DBI::SQL::Nano, AnyDBM_File, DB_File, BerkeleyDB, MLDBM, YAML::MLDBM MLDBM::Serializer::JSON